Supporting dynamic pipeline changes using Class-Based Object Versioning in Astro-WISE

Mwebaze, Johnson; Boxhoorn, Danny; Rai, Idris; Valentijn, Edwin A.

Published in:
Experimental Astronomy

DOI:
10.1007/s10686-011-9270-1

Publication date:
2013

Exp Astron (2013) 35:157–186
DOI 10.1007/s10686-011-9270-1

ORIGINAL ARTICLE

Supporting dynamic pipeline changes using Class-Based Object Versioning in Astro-WISE

Johnson Mwebaze · Danny Boxhoorn · Idris Rai · Edwin A. Valentijn

Received: 20 June 2011 / Accepted: 2 November 2011 / Published online: 3 December 2011

Abstract Understanding the difference between data objects is a major problem especially in a scientific collaboration which allows scientists to collectively reuse data, modify and adapt scripts developed by their peers to process data while publishing the results to a centralized data store. Although data provenance has been significantly studied to address the origins of a data item, it does not however address changes made to the source code. Systems often appear as a large number of modules each containing hundreds of lines of code. It is, in general, not obvious which parts of source code contributed to the change in data object. The paper introduces the Class-Based Object Versioning framework, which overcomes some of the shortcomings of popular versioning systems (e.g. CVS, SVN) in maintaining data and code provenance information in scientific computing environments. The framework automatically identifies and captures useful fine-grained changes in the data and code of scripts that perform scientific experiments so that important information about intermediate stages (i.e. unrecorded changes in experiment parameters and procedures) can be identified and analyzed. The benefits of such a system include querying specific methods and code attributes for data items of interest, finding missing gaps of data lineage and implicit storage of intermediate data.

Keywords Scientific computing · Object versioning · Data provenance

J. Mwebaze (✉) · D. Boxhoorn · E. A. Valentijn
Kapteyn Astronomical Institute, University of Groningen, Landleven 12,
9700 AV Groningen, The Netherlands
e-mail: jmwebaze@cit.ac.ug

I. Rai
College of Computing and Information Sciences, Makerere University,
P.O. Box 7062, Kampala, Uganda

1 Introduction

Astronomers, like any other scientists often prototype data analysis scripts using high-level languages like `Python`. These scripts parse input, execute a sequence of steps and create output. A script could be a class, method or a function or lines of code as defined in the programming language being used. Developed scripts live in conventional file systems like in centralized versioning system (e.g. CVS[1]), or in a personalized checkout on a folder on the researchers computer.

Being able to quickly implement and refine prototype code is vital to the research process, since the specifications for research code (as compared to production-quality code) are often ill-defined and constantly-changing, fairly distributed, are made of thousands modules that evolve rapidly and independently.

A typical data analysis work cycle is a recursive process that consists of basic actions: create an analysis script (or pipeline) and execute its initial run; view results. If not satisfactory, modify input data or an analysis function (script), or add an analysis function and re-execute the pipeline; while publishing results to a centralized data store.

During a data analysis work cycle not all intermediate or working versions/variations of scripts become part of the software repository and therefore not all versions/variations of classes during the scientific analysis process will necessarily be released and made part of the software repository. The way people use versioning systems is by committing changes from time to time when they feel that they have completed a feature, but what happens between two commits is a mystery. For example, a user could make a change to a script, process and store a result A, makes another change, processes and stores the result (A'). Both A and A' are committed into a centralized database but we do not know the difference between A and A'. The user could possibly commit his code after making and processing several other results. We can only make some hypothesis about the difference between A and A' by considering the two successive states of the script at hand, which could lead to a false negatives.

Often scientists may prefer to use their own implementation of a script and probably maintain other variations of the same implementation for comparison purposes. A study carried out on scientists computers also revealed that the users maintained several copies of a script saved under different names. (e.g. script1.py, script2.py, script3.py, etc.). This is an indication of conflicting or alternative implementations. If all these working versions of classes from different users become part of the repository, these differences (conflicts) could be detectable, but perhaps hard to be resolved.

A scientist faces two principal obstacles when working with data: firstly, understanding the origins of data (i.e., *data provenance* [9], and secondly, understanding the differences between two data items (i.e., *object versioning*).

[1] Concurrent Versioning System http://www.cvshome.org/

Provenance systems (e.g., StarFlow [7], Chimera [8], Kepler [1], Karma [19], and ZOOM [4]) do trace lineage by capturing and storing a complete trace of a data flow. However changes made to the scripts are usually ignored. At a coarse level, provenance systems refer to `'a procedure x'` was run on `'data y'` [14] but at the lowest, scientists are also interested in knowing what changes in `'procedure x'` made data to appear the way it is. Capturing these kind of changes is possible when all users sign up to a systematic approach of change management, processing and storage. However this is not often the case. For example, `Astro-WISE` [3, 20] provides scientists with an Astro-WISE environment and allows users to have their own code. Users can modify code in their repository to evaluate their specific questions about the datasets, change/apply their own algorithms on datasets and derive results following their insights. For each object processed using Astro-WISE, we are able to keep truck of all dependencies, and processing parameters. However, tracing what users are doing with the code in their own repositories and how the code is affecting published data is still a challenge. Knowing the changes done to script and the data created, is of utmost importance especially in scientific collaboration which allows scientists to collectively reuse data, modify and adapt scripts developed by their peers to process data while publishing the results to a centralized data store.

To support scientific collaboration, we propose a mechanism that allows collaborators to work on source code, process data while we automatically capture relevant changes to source code and link the changes to data objects created. With this mechanism we do not automatically merge changes like version control systems. Rather, we create persistent code objects in a database representing source code and the changes made to source code.

In order to support the proposed mechanism we require; a versioning mechanism because each user needs to know how their collaborators' work relates to their own or how code objects are related, a centralized repository to manage all the code objects and to provide the means for notifying collaborators when changes occur and a linking mechanism to link versions to data objects. Most scientific environments depend on the versioning capabilities available in versioning systems (e.g. CVS, SVN)to keep a log on how source code has changed over time. However, these tools are limited in their ability to detect differences in programs because they provide purely textual differences [16]. These tools do not consider changes in program behavior, they are based on copy-modify-merge model, therefore no support for alternative implementations, they can not group related changes, so that they appear as a single logical entity.

The level granularity of all of the commonly used versioning systems are file and/or program based and to some extent lines. While files/programs are too coarse-grained for detailed analysis, lines seem to be too fine-grained. A line is usually too excessively coupled to other lines (commands) to be considered a unit of comparison. For example, SVN's revision numbers apply to entire trees not individual files. If we version an object based on a build/release number (e.g., from Maven, SVN) then the version number that applies to the whole

program will be the version of all objects even when the classes used to make a specific object may not have changed. Moreover, the version numbers created from such tools are purely opaque identifiers because a simple textual, space change in such tools will create a new version.

Using this versioning model, browsing information in a precise way for such systems is not possible as classes themselves and even methods, are not part of the navigation. The versioning information considered by these tools is the number of files, directories (and other dependencies) and their relationships, as well as lines added, deleted or modified for each commit(version). This might be helpful to a software developer, but not a scientist who is trying to understand the differences in his data to establish relevant links between data and changes in the program elements. There is also significant work that has been done on detecting differences in source code in literature however, these techniques are based on purely syntactic differences [11, 13, 21] at the same time, the results of these tools are targeted towards software maintenance tasks.

The Class-Based Object Versioning (COVA) we propose here will version data/source code, in much finer-grained ways. The versioning will be based on program entities (e.g., classes, methods and class hierarchies relationships) while linking this information to the data objects. This information will be made persistent in the database which can then be queried directly, without needing costly parsing steps. Scientists can then focus on more precise relationships and exploit the relationships to evaluate the variations of the lines of code (e.g., of a single method) and how the variation affects the data. We would like at the lowest level to link the changes made to program entities that do change results of a class to the data objects created. If two data objects were created by different variations (versions) of the same class, the difference between the two objects can be explained. This paper makes the following contributions;

- a framework that supports automatic change capture which allows scientists to define new processing routines, support multiple variations of methods for solving a task and let users choose during run-time, which of the different variations is most appropriate for them.
- develop framework that identifies and classifies changes based at different abstraction levels using object-oriented structures while linking the changes to derived data.
- an extensible infrastructure that will allow effective sharing of data and source code and provide common resource discovery mechanisms, by representing source code and extracted attributes from source code as persistent objects.

The rest of the paper is organized as follows; We briefly review Astro-WISE in Section 2 and present the underlying design objectives of COVA in Section 3. In Section 4 we present the framework for change detection. We present our object linking and version management in Section 5. We evaluate

this work in Section 6. We review related work in Section 7 and Section 8 concludes and presents an outline of future work.

2 Astro-WISE review

The work we present in this paper is based on Astro-WISE. Readers are referred to [3, 20] for detailed information on Astro-WISE. Each Astro-WISE's user has an Astro-WISE environment from which the user interacts with the system and processes data. Included in the environment is the complete source code for Astro-WISE. This environment allows the astronomer to plan, modify and rerun the analysis process to fit the particular needs that follows from the astronomical questions posed to the data. This often requires that a user modify source code and process data as needed.

Although we have all data stored in the system, we do not have a connection between the class/methods that created the data. Classes in Astro-WISE are associated with the various conventional calibration images, data images, and other derived data products. For example, bias exposures become instances of the `RawBiasFrame` class, and twilight (sky) flats become instances of the `RawTwilightFlatFrame` class. From this association we know the class that was used to make a derived data product, however the specific version of the class is probably in an individual's code base. If significant changes were made to the class, retrieving the object without access to the specific version of the class that used to make it, may throw an exception. Since most of the changes remain on the local node, it becomes very hard other researchers to learn from another user, or more still understand the differences between data objects.

Our goal in this work is to capture fine-grained code and data provenance for each user, by capturing and analyzing detailed actions performed on source code that exists in the users checkout repositories. We then present this information in a way that will enable collaboration such that other astronomers could benefit from results and methods used by their peers.

3 Architecture

To address the current inherit limitations of Astro-WISE as highlighted in Sections 1 and 2, we need a versioning mechanism to detect changes that go beyond simply providing purely textual differences, an object linking mechanism that links versions of classes to derived data objects and a centralized repository to manage all source code and derived data objects. The combination of these features will provide common source code/data discovery mechanisms and allow effective sharing of source code, methods and data.

Because we expect to encounter a large number of code edits, it will be inefficient to create a new version of a class for each change detected. A code edit can be lexical, syntactical or semantic. For this work we only consider changes that would change the results of a class. The requirement is that, given

a class with same input data and arguments, the class should always return the same derived object. Source edits that change spacing, comments, and other minor cosmetic tweaks that do not change a result of a function are ignored. In general, this technique works as follows:

1. The scientist runs a script in Astro-WISE environment. If this is the first time this script has been run, a request to save the computed results to the database, will automatically create a code object. A code object(code object) is persistent object that stores information extracted from source code. e.g., classes, methods, class attributed and variables, class dependency relationships, in a database while linking the complete source code file with this information. The persistent code object will have a version number which is used to reference the code object to the derived object.
2. During subsequent runs of the same script (possibly after some edits), Astro-WISE compares the current script with the source code file linked to the persistent code object to detect changes. This comparison is only done when a user requests to commit the results.
3. If changes are detected Astro-WISE automatically creates a new version of the code object and creates dependencies between the code object, the created data objects and the source code edits.

3.1 Class-Based Object Versioning (COVA)

This section gives an overview of Class-Based Object Versioning, it describes the relationships among classes and explains what kind of dynamic information is used as input to Class-Based Object Versioning.

3.1.1 Versioning units and object versions

An important aspect of the Class-Based Object Versioning is how to select the elements that shall be compared. These elements are the versioning units that form the basis of comparison. We define versioning unit as an element associated to versioning information. A new version of the element is created when any part of it is modified.

Each versioning unit may have its own version but the aggregation of these versioning units (the class) may have another version. i.e., a version number attached to an object should have the state of all classes, methods and attributes that were used at the time of making the object. For example, a `Python` class is composed of methods and attributes (locally defined in the class and inherited). In this scenario, a class is a versioning unit and its methods and attributes are also versioning units. If a method or an inherited class is changed, a new version of the class is also created, because the class has been indirectly changed as shown in Fig. 1. Therefore the object processed with this modified class will have a new version, even when the most specific class may not have changed.

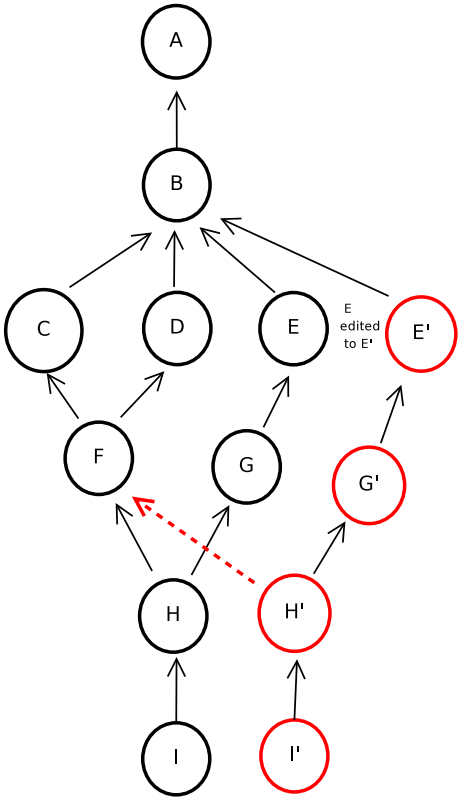


Fig. 1 The graphs shows the dependency hierarchy of class I. In the graph class E was modified to a new version E'. Since G, H and I are dependent on E, new versions of G, H and I will be created as G', H' and I' respectively. Notice that H' still refers to F

3.1.2 Selecting versioning units

For object-oriented systems, the state of an object depends not only on its own data attributes but also on the objects it refers to. This means, the class that creates an object may use services of other classes to create the object. In such cases, a class will have elements to be versioned which are distributed through different files. Due to the complexity of this data model, it is not desirable to have a single versioning behavior for all objects. This is the reason why version numbers from versioning control systems e.g., SVN or version numbers from build tools like Maven, would fail to address the versioning requirements for this work. Current versioning control systems do not work with fine grained versioning information. Our approach allows a fine-grained definition of versioning units. For example, a class may be defined as an atomic versioning unit and also its methods, and attributes as other versioning units.

We use the examples in Figs. 2 and 3 to describe intuitively the selection of the versioning units. Since this framework considers only changes that have an effect on data, which are determined through testing (see Section 4.1.2), it is of paramount importance that we carefully select the versioning units, to reduce on the amount of time that will be required for testing.

Fig. 2 Example source code R

```
1   class BaseFrame(DBObject):
2       def load_image(self):
3           pass
4       def load_header(self):
5           pass
6       def empty_header(self):
7           self.header = darma.header()
8
9   class ReducedScienceFrame(BaseFrame):
10       def debias(self):
11          self.bias.load_image()
12          self.image -= self.bias.image
13       def flatfield(self):
14           pass
15
16
17       def update_astrom(self):
18          self.load_header()
19       def make(self):
20         self.debias()
21         self.flatfield()
```

A class-hierarchy graph is a straight forward relationship between classes. Class-hierarchy changes may affect calls to methods in any classes in the hierarchy. A straight forward approach for selecting versioning units, is to

Fig. 3 Modified source code for R, R'

```
1   class BaseFrame(DBObject):
2       def load_image(self):
3           pass
4       def load_header(self):
5           pass
6       def empty_header(self):
7           self.header = eclipse.header()
8
9   class ReducedScienceFrame(BaseFrame):
10       def debias(self):
11          self.bias.load_image()
12          self.image += self.bias.image
13       def flatfield(self):
14           pass
15       def load_header(self):
16           pass
17       def update_astrom(self):
18          self.load_header()
19       def make(self):
20         self.debias()
21         self.flatfield()
```

consider all classes and methods in a class-hierarchy. From our example (Figs. 2 and 3), if the `ReducedScienceFrame` class is the most specific class, then classes `ReducedScienceFrame` (R), `BaseFrame` (BF), `DBObject`(DBO) and their methods would all be considered as versioning units. All these versioning units would be compared to find differences between R and R', where R' is the modified version of R.

Assume that the change at line 12 in Fig. 3 is the only change between R and R'. The change detection algorithm should indicate that the R class has changed and specifically that the debias method has changed. In this case the debias method will have a new version and the R class will have a new version. Lets also consider the addition of the load_header (line 15 and 16) method in Fig. 3 . This change leads to a possibly different behavior for all statements that may call load_header method. A simple difference operation for the R class will pick up load_header as a new method in R' but not in R. This is not a new method but a case of method overriding. Calls to load_header in R will be bound to the load_header of BF class, whereas calls to load_header in R' will be bound to the load_header in R'. During change detection, our comparison should compare the load_header form the BF class to the load_header of R' class.

Notice from Figs. 2 and 3, the BF class defines three methods, `load_header`, `load_image` and `empty_header`. The analysis of the BF class shows that the `empty_header` methods has been modified. Since R is a subclass of the BF class, we can say that the R class has been changed indirectly. Note however that this change does not affect the R class since `empty_header` is never called in the R class. In such a case, the `empty_header` method should not be considered as a versioning unit at-least for the R class.

Although the R class defines several methods, not all methods might be called during a data-flow. This could be caused by conditional statements or specific input parameters which might trigger specific methods to be called. `Python` also defines the `getattr` method which can be used to call methods when a condition is True/False. For example `result = obj.empty_header(args)` is equivalent to `result = getattr(obj, "empty_header")(args)`. In this example, since `empty_header` method has been passed to `getattr` as a string, a static analysis of code, would not detect this kind of method call. In the same way `getattr` supports dynamic binding in python.

Using static analysis of code, the versioning units as described above can be extracted. However, static analysis is inherently more difficult for object-oriented languages than for procedural languages. Static analyses need to make conservative assumptions in the presence of dynamic binding, which weaken the precision data collected. We therefore use dynamic information to extract the required information where static analysis fails. However, we need to note that dynamic information depends upon suitable input data and the test environment in which the program are executed, dynamic information is therefore used as a guide. Specifically, we use dynamic analysis, using python

`settrace`, to record the execution profile (Control-Flow Information) of all method calls of a given program run. This execution profile also specifies the order of execution. From the execution profile, we gather the fact that a method has been executed at least once. To get the specific versioning units we mutually intersect the execution profile with the statically obtained information.

Algorithm 1 Selecting Versioning Unit for a class

Input: `ClassName`, source code, Control-Flow Information (CFI)
Output: Versioning Units (VU)
1: Called Methods(CM) $= \varnothing$
2: Defined methods (DM) $= \varnothing$
3: Inherited Methods (IM) $= \varnothing$
4: Attribute (A) $= \varnothing$
5: $VU = \varnothing$
6: tree $\leftarrow$ `ast.parse(source code)`
7: **for** node $\in$ tree **do**
8: **if** node (n) is `ClassDef` node and node name is `ClassName` **then**
9: tree = node
10: $VU = VU \cup \{n\}$
11: **end if**
12: **end for**
13: **for** node (n) in tree **do**
14: **if** node is Call node **then**
15: $CM = CM \cup \{n\}$
16: **else if** node is `FunctionDef` node **then**
17: $DM = DM \cup \{n\}$
18: **else if** node is `Attribute` node **then**
19: $VU = VU \cup n$
20: **end if**
21: **end for**
22: $VU = VU \cup CM$
23: $VU = VU \cup (CFI \cap (DM - CM))$
24: $IM = CM - DM$

Algorithm 1, is used to parse a class and select the versioning units. We use Abstract Syntax Tree (AST) framework to parse code into a tree then walk the tree to select appropriate versioning units. Line 5 in the Algorithm 1 parses provided source code into an AST. The provided source code could have implemented several classes. Line 7 selects the class to be versioned and makes it the current tree in line 8. The class is added to the list of versioning units in line 9. We visit each node in the current tree while checking if the node is callable/Called Method (line 13 & 14) or a method definition/defined method (line 15 & 16). A called method is considered Versioning Unit (line 19). The difference between defined and called methods, should be the methods which were not called. However to cater for effects of dynamic binding, we find out if these methods are not part of the control-flow information (Line 20). Those that are part of the control-flow information are added to the versioning units in line 20. We also have inherited methods (Line 21), which are those methods that were called in a class, but were never defined in that class.

We now translate Algorithm 1 to an iterative algorithm that follows the class-hierarchy graph to detect versioning units for each class in the graph. However for each parent classes, we are only interested in capturing methods which exist in set IM. So after the analysis of all called and defined methods, only those elements that will be in set IM will be compared.

3.1.3 Class construction

Since each versioning unit is versioned separately, this gives users the power to construct classes on-the-fly during processing by putting together the specific versions of the versioning units that make up a class. For example, consider the process of creating a `ReducedScienceFrame` object. This process involves the following operations;

1. overscan correction and trimming
2. subtraction of the BiasFrame
3. division by the MasterFlatFrame
4. scaling and subtraction of a FringeFrame if indicated
5. multiplication by an IlluminationCorrectionFrame
6. creation of the individual weight image
7. computation of the image statistics

The process can be represented as an evaluation graph as shown in Fig. 4. The circles represent input/output data, the rectangles represent operations

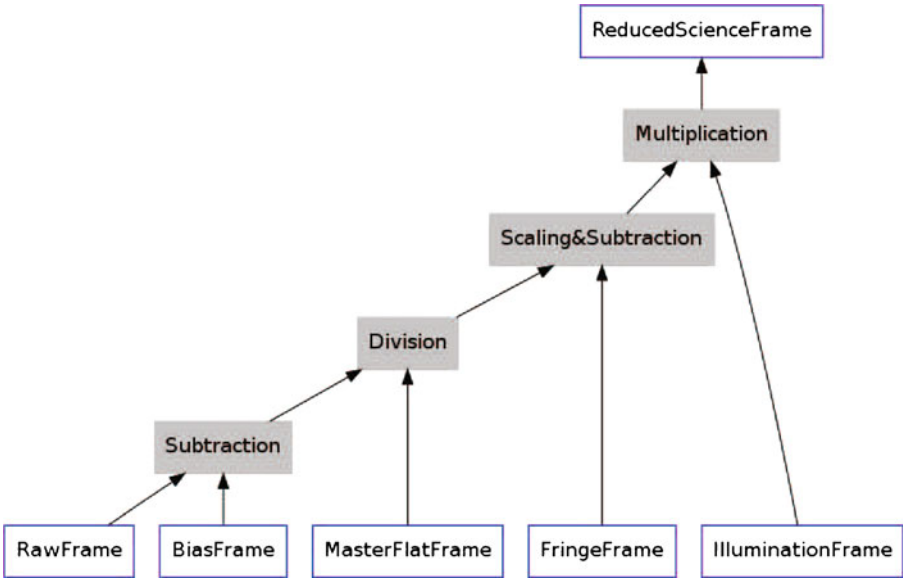


Fig. 4 An evaluation graph: the nodes in the graph are connected with edges that show the flow of processing. *Gray nodes* represent persistent operations. e.g., a method/class/or module, while the *blue nodes* represent input/output to the operations

and the arrows show the direction of the dataflow. If each operation (An operation may be implemented as a class or as method) is a versioning unit, then a user needs to select specific versions of each operation, construct the whole pipeline for making the `ReducedScienceFrame` object. The final version of the `ReducedScienceFrame` class that was used to make the `ReducedScienceFrame` object is computed from the versions of each of the operations used during the making of the object. After processing the `ReducedScienceFrame` object, a new version of the class is created and linked to the object.

3.2 Centralized repository

In order to efficiently capture and link code edits to data objects created, we use a relational database management system (RDBMS) for our centralized repository. We choose to use a RDBMS because these systems provide secure access protocols, support concurrent transactions from multiple users, and include trigger mechanisms for alerting users when the database is updated.

3.3 Persistent code objects/object versioning

Code objects are persistent objects that store information extracted from source code. e.g., classes, methods, class dependency relationships, in a database while linking the complete source code file with this information. Classes and methods are versioned separately, therefore methods are also represented as objects.

Each versioned class has an associated code object from which the versions of the derived objects are obtained. Each code object knows all its dependencies (methods and relationships with other code objects) and their states (versions). Each code object is linked to other code objects which can themselves be linked to other code objects. A code object can be visualized as a version tree, where each node corresponds to a versioning unit and each edge corresponds to a relationship to other versioning units. Each version tree will be same for all derived objects of same version. Because the version tree captures that state of each versioning unit, users have great flexibility of comparing different versions of code objects. Users can easily see how their collaborators have taken different approaches to solving related problems and how their techniques relate or defer from their own ideas. A user can construct a class on the fly as described in Section 3.1.3.

For each data (or derived) object processed, a persistent link to the version of code object that was used to make the object is created and is stored as the part of the object's attributes (or metadata). This ensures that during the object's de-serialization, that appropriate classes are called to reconstruct the object. The code object is identified by a unique object identifier (*object_id*) and version number *version_no*. The *object_id* and the *version_no* of a code object are used as reference to identify the relationship between the source code and the data/derived object.

3.4 Potential benefits of COVA

3.4.1 OOP and persistent objects

If source code is represented as objects, we can use the power of object-oriented techniques to analyze and perform useful queries on these objects. Rather than viewing source code as linear streams of ASCII characters, we can now view source code as objects (code objects). It will certainly allow extreme data validation, data reuse, ability to view it in many ways, and to search for data or code by any attribute/key.

We can associate temporal information with individual code objects. This provides means of recording code object histories and thereby allowing the histories of code object and the types of code object to be easily traced and compared. To show the potential benefit of this approach, potential queries as below can be answered;

- I want to apply an astronomical analysis program to millions of objects. If the program has already been run and the results stored, I will save weeks of computation
- I want to find out and recompute all products made from a method that implements the `x, y, z` attributes and where `y='some value'`.
- I have detected a bug in a program, I want to know which derived data to recompute
- A user has presented two final re-gridded images, I would like to know the difference between the two images.

3.4.2 Code-based searches and data lineage

Code-based searches should enable users find data of interest based on code/attributes/changes that was used to make the objects. For example, one could search for all data created by a particular method implementation. Combining code-based searches and data lineage (provenance) introduces functionality not available in existing systems. For example, suppose objects were made with a version of code with a bug. A code based search should be able to identify all objects processed with this code. Then data lineage is used to find all derived objects that depend on the data that was created with code with a bug and also objects created prior to version with the bug. These two traces would precisely identify the appropriate objects and versions that need to be recomputed (or removed).

3.4.3 Incremental re-computation

Computational scientists often speed up execution times by refactoring their scripts into stages (separate modules) while saving intermediate results to avoid recomputing them in subsequent runs. This eliminates processing steps when intermediate data products have already been generated (by another

workflow or previous execution of this workflow). Selection of intermediate data is based on provenance information [6, 15]. Specifically if the input and processing parameters have not changed that are required to process intermediate data for the current process.

However, when source code changes, provenance information used to determine if the intermediate data should be re-used is no longer sufficient, since some changes in source code might produce different results. This work adds another dimension to dependency checking when selecting intermediate data products for re-use. Specifically we compare source code currently in use and source code that was used to process the intermediate results. If the two do not match, then re-computation of intermediate data will be necessary.

3.4.4 Management of intermediate data

Common to the dataflow programming frameworks, is the existence of intermediate data produced as an output from one stage and used as an input for the next stage during a dataflow. Intermediate data is short-lived, used immediately, written once and probably read once. Intermediate data management problem is largely unexplored in current dataflow programming frameworks. However due to the requirements of storing and capturing provenance for eScience and the evolving nature of eScience, scientists have resulted into the storage of intermediate data explicitly.

With this framework, rather than storing intermediate data explicitly, we instead store a link to the code object that was used to make the intermediate data. Intermediate data is only stored if it is necessary for performance reasons and the results can be shared between pipelines. This ensures that the appropriate version of the intermediate data product that was used during a processing can be re-created on-the-fly when required.

4 Change detection

To determine if a new version of a class has to be created, two classes are matched together to find the differences. We use two metrics to determine a matching between two classes. The first metric is the semantic difference in the class and the second metric is if the changes identified affect the results of a computation. We also acknowledge that changes to the methods which speed up execution times are very important when processing large data-sets, however in this framework such changes have not been considered and are only limited to the current versioning mechanisms in Astro-WISE.

For each class, we generate and build a dependency graph (G). We denote a dependency graph of a class as node-labeled directed graph $G(V, E)$ where $V(G)$ denotes the set of all nodes in G and $E(G)$ denote the set of all edges in G. The graph contains class and method nodes. A class node is connected to

the method node by an edge. For a derived class, there exists an edge between the class and the classes from which it inherits.

The dependency graph represents the class and its interaction with other classes. This graph accounts for effects of inheritance, scoping, polymorphism and dynamic binding as demonstrated in Section 3.1. What remains is to find the difference between the graph generated for the original class (G) and the graph of the modified class (G').

Non-leaf nodes in the graph are the classes, and leaf nodes are the methods and attributes of each class. We begin by matching the classes, then for each class matched we match the methods and attributes. We do not build enhanced control-flow graph to compare methods as work done in [2], we instead compare `Python` bytecodes. We do so because we want to avoid source edits that change spacing, comments, and other minor cosmetic tweaks that do not alter a function's behavior. If changes are detected during method comparison, we continue to check if the results of the two methods differ. i.e., given two methods M and M' and same input dataset o, if $M(o) \neq M'(o)$, then we assume the changes made to M to create M', are significant to create a new version of a method, and eventually a new version of a class. The actual differences between the two methods are determined by comparing their disassembled bytecodes.

All attributes in Astro-WISE are persistent. Constants are excluded from source-code and are also represented as a persistent attributes at class level. So changing the a value of a constant will be a modification on the data object and not source-code. Therefore the graph includes attribute nodes and an edge connecting a persistent attributes node to the class where the attribute is defined.

4.1 Class matching

Graph (class) matching measures the similarity between two graphs using the notion of graph edit distance, i.e., it produces a set of edit operations that model inconsistencies by transforming one graph into another [5]. Typical graph edit operations include the deletion, insertion and substitution of nodes and edges. Each edit operation is assigned a cost. Then the edit distance of two graphs G and G' is found by searching for the sequence of edit operations with the minimum cost that transform G into G'. A similar problem formulation can be used for trees.

Our interest is not the computation of the edit distance, since some changes will eventually be ignored if they do not have an effect on the data. We therefore provide a new graph representation and a differencing algorithm that will identify and classify changes between two graphs corresponding to a class while comparing known results of changed methods to verify the effect of the changes on the methods. Our goal is to enable data re-use and foster collaboration between scientists. Rather than computing the edit-distance, we associate the detected changes between classes and/or methods

to the derived data objects that shall be created through the modified classes.

4.1.1 Node matching

To find differences between two graphs, we carry out a matching between corresponding nodes. We begin by matching non-leaf nodes (class nodes) and then match the leaf nodes. For two graphs G and G'. Leaf nodes that appear in G that do not appear in G' are deleted classes, whereas leaf nodes in G' and not in G are added classes. The same applies to non-leaf nodes.

To ensure uniqueness of a node, we introduce the *pathToNode* for this purpose which is defined as follows;

Let V denote a set of all nodes in Tree (T), if $v \in V$, the $pathToNode(v_n) = v_1.v_2 \ldots v_{n-1}.v_n$, where v_1 is the root of T, and $v_1, \ldots v_{n_1}, v_n$ is the path from root v_1 to v_n. The '.' represents a link property attribute which defines relationships between two nodes that are transitively connected. Based on the definition of the *pathToNode*, we define a *matching* M as below;

Given set of node pairs (v_a, v_b) where $v_a \in V_a$ and $v_b \in V_b$, M is called a matching from T_a to T_b, *iff*

1. $v_a, v_b \in M$, $v_a \in V_a$, $v_b \in V_b$, $\text{pathToNode}(v_a) = \text{pathToNode}(v_b)$
2. $\forall(v_{a1}, v_{b1}) \in M$ and $(v_{a2}, v_{b2}) \in M$, $v_{a1} = v_{a2}$ *iff* $v_{b1} = v_{b2}$
3. Given $(v_a, v_b) \in M$, suppose v'_a is a parent of v_a and v'_b is the parent of v_b, then $(v'_a, v'_b) \in M$
4. $\text{bytecode}(v_a) == \text{bytecode}(v_b)$

We now use the *pathToNode* and *matching* definitions to recursively match nodes in G and G'. We begin the comparison at the class level. After matching classes we then match methods for each pair of unmatched classes. Unmatched classes are those classes that have differences in their implementation. For any unmatched methods, we compare the output of two methods and we continue to log the semantic differences if the output of the methods differ. This process is summarized in Algorithm 2.

4.1.2 Comparing output of two methods

Given the same list of source code segments, processing environment and arguments, a compilation should always return the same derived object. If the results defer, then we can confirm a change in the implementation. We note that when external imports e.g., `numpy`, `pyfits` are modified, these too can change a result of a method. To diminish the effects of such changes, the test environment includes a standard setup with known versions for external exports and expected output from each method. The modified methods are plugged into this test environment while other dependencies remain constant. We only execute sections that have been modified.

Algorithm 2 Change Detection

Input: Original Class C, Modified class C'
Output: set of unmatched methods classes, C'' unmatched methods M'', set of semantic differences $DIFF$

```
Parse classes C and C' into their dependency graphs G and G' respectively
T ← V(G) Set of all none leaf nodes of G
T' ← V(G') Set of all none leaf nodes of G'
for each node in t ∈ T and t' ∈ T' do
    matched ← match(t, t')
    if matched then
        t, t' are equivalent
        continue
    else
        V ← V(t) Set of all leaf nodes of t
        V' ← V(t') Set of all leaf nodes of t'
        for every node m in V do
            for every node m' in V' do
                matched ← match(m, m')
                if not matched then
                    compare the output of both methods given the same input data
                    if output differ then
                        add m, m' to M'', remove m and m' from V and V' respectively
                        add semantic differences between m and m' to DIFF
                    else
                        remove m and m' from V and V' respectively
                    end if
                else
                    remove m and m' from V and V' respectively
                end if
            end for
        end for
        if M'' is not empty, add t, t' to C''
        remaining in V and V' are new nodes(methods)
    end if
end for
```

If a method M was modified to M', we check to see if for any object o, $o.M() == o.M'()$. If N, and M are two methods, where N precedes M during execution, if the state of object o, at the time the compiler completes the execution of N is o' i.e., $o.N() = o'$. and the $o'.M() = o''$ then if $o'.M' \neq o''$ then we can confirm that changes in M, are significant to create a new version of M i.e., M'.

4.1.3 Semantic differences between methods

To get the semantic differences between methods that have not been matched, we use the `Python dis` module to disassemble the bytecodes of the unmatched methods. Figs. 5 and 6 shows the disassembly of two methods. The methods in Fig. 5 are the original methods while the methods in Fig. 6 are the modified versions. Line 6 and 17 are the changed lines. In the method `debias` the operation between attributes `self.image` and `self.bias.image` was `INPLACE_SUBTRACT`, that has been changed to `INPLACE_ADD` in the modified method.

Fig. 5 Disassembled bytecode of __init__ and debias methods

```
1   Disassembly of __init__:
2                0 LOAD_FAST 1 (image)
3                3 LOAD_FAST 0 (self)
4                6 STORE_ATTR 0 (image)
5
6               27 LOAD_CONST 1 (0)
7               30 LOAD_FAST 0 (self)
8               33 STORE_ATTR 3 (commit)
9
10   Disassembly of debias:
11              10 LOAD_FAST 0 (self)
12              13 DUP_TOP
13              14 LOAD_ATTR 1 (image)
14              17 LOAD_FAST 0 (self)
15              20 LOAD_ATTR 2 (bias)
16              23 LOAD_ATTR 1 (image)
17              26 INPLACE_SUBTRACT
18              27 ROT_TWO
19              28 STORE_ATTR 1 (image)
```

5 Object linking and version management

We describe in this section Astro-WISE's mechanisms for linking objects and how we manage versions between objects. We have used persistent code objects that represent a set of closely related parameters extracted from a class that are represented as an object stored in a relational database.

Fig. 6 Disassembled bytecode of the modified __ init__ and debias methods

```
1   Disassembly of __init__:
2                0 LOAD_FAST 1 (image)
3                3 LOAD_FAST 0 (self)
4                6 STORE_ATTR 0 (image)
5
6               27 LOAD_CONST 1 (1)
7               30 LOAD_FAST 0 (self)
8               33 STORE_ATTR 3 (commit)
9
10  Disassembly of debias:
11              10 LOAD_FAST 0 (self)
12              13 DUP_TOP
13              14 LOAD_ATTR 1 (image)
14              17 LOAD_FAST 0 (self)
15              20 LOAD_ATTR 2 (bias)
16              23 LOAD_ATTR 1 (image)
17              26 INPLACE_ADD
18              27 ROT_TWO
19              28 STORE_ATTR 1 (image)
```

5.1 Persistent and code objects

The persistent object hierarchy makes the core of this framework. We automatically make all objects persistent in the database as attributes, as fully integrated objects or as descriptors. Source-code files are not stored in the database, however their unique filenames and links to their code objects are stored in the database.

5.1.1 Definition of persistent attributes

All Astro-WISE classes are derived from a customized metaclass. Using the metaclass we can manipulate class creation, object instantiation and method execution. We defined another class `DBObject` derived from the metaclass which is the root class of the hierarchy of the persistent classes. This class defines the primary key (i.e., `object_id`) of all objects. Any class that inherits `DBObject` automatically becomes persistent. This creates all the necessary schema structures, such that attributes and data created or used during the processing will be stored. Likewise, a persistent attribute is defined by using the following expression in the class definition.

```
prop_name = persistent(prop_docs, prop_type,
                       prop_default)
```

where, `prop_name` is the name of the persistent property, and `persistent` is constructed using three arguments: the property documentation, the type of the property, and the default value for the property respectively.We distinguish between 5 different types of persistent properties, based on the signature of the arguments to persistent.

- **descriptors**: If the type of the persistent property is a basic (built-in) type, then we call the persistent property a descriptor. Valid types are: integers (int), floating point numbers (float), date-time objects (datetime), and strings (str).
- **descriptor lists**: Persistent properties can also be homogeneous variable length arrays of basic built in types, called descriptor lists. Valid types are the same as those for descriptors. Descriptor lists are distinguished from descriptors by the property default. If the default is a Python list, then the property is descriptor list, else it is a simple descriptor.
- **links**: Persistent objects can refer to other persistent objects. The corresponding properties are called links. If the type of the persistent property is a subclass of `DBObject`, then the property is a link.
- **link lists**: Persistent properties can also refer to arrays of persistent objects, in which case they are called link lists. Link lists are distinguished from links by the property default. If the default is a Python list, then the property is link list.

– **self-links**: A special case of links are links to other objects of the same type. These are called self-links. If no type and default are specified for the call to persistent, then the property is a self-link.

For example, consider the following persistent attribute definition

```
attr_1 = persistent(' ', ClassA, (ClassA, (),{})
attr_2 = persistent(' ', ClassA, None)
attr_3 = persistent(' ', ClassA, [])
attr_4 = persistent('Link to object of ClassB')
```

then,

– `attr_1` is a link to an instance of `ClassA`
– `attr_2` is a link to an instance of `ClassA`,:
– `attr_3` is a array of links to instances of `ClassA` (default empty)
– `attr_4` is a link to another instance of `ClassB`.

An attribute defined as `filename = persistent('File part', str, ' ')` supports the storage of files by creating persistent link to the source code file. Note that persistent properties are inherited.

A class (or derived object) is linked to the code object through a persistent attribute called `code_object`. as shown below;

Example 1 Linking an object to the code object

```
class ReducedScienceFrame(DBObject):
   code_object = persistent('The CodeObject used to make this object
   [None]', CodeObject, None)
```

5.2 Version control

Our requirement is that we are able to test the equivalence of two code objects. Each code object has a unique identity, a version attribute and a version predecessor attribute. If a code object is created as a new version of another existing code object, its version predecessor pointer points to its version predecessor otherwise its version predecessor pointer is null. Each code object is read-only. Changes made to an existing code object are stored as a new version of the code object.

We have created versioned groups called 'type' which are identified by the class names. For each type, the version number counter begins from 1 and incremented by 1. For example, the type of a class called `BaseFrame` will be BaseFrame.

Three basic operations for version control are new, edit and delete. Each of these operations is modeled as a function that takes a class as its major input and returns a new version and a code object stored in the database.

The `New` operation creates and adds new code object to a database. The code object represents new type which is different from any existing type in the database. This new code object will have no version predecessor.

Algorithm 3 New Operation

```
Input: class
Output: code object
  type ← get-type of class
  if type exists in database then
    return
  else
    create new code object
    set version number of to code object 1
    set version predecessor pointer to NULL
  end if
```

The `Edit` operation is used to create a new (edited) version of an existing type in the database. This operation takes as input a class that is assumed to be modified and matches the class against an existing code object. Based on the results of matching, a new version of a code object might be created. If a new version of the code object is created, the version predecessor attribute of the new version will point to the code object that was used during the matching. A user has an option of selecting a specific version of the code object to use during matching, otherwise the latest version will be used during the matching process.

Algorithm 4 Edit Operation

```
Input: class, version_number
Output: code object
  modifiedClass ← class
  type ← get type of class
  if type exists in database then
    max_version ← get max_version_number of type
    if version_number is provided then
      vers ← version_number
    else
      vers ← max_version
    end if
    originalClass ← retrieve source code for code object with vers
    match (modifiedClass, originalClass)
    if not matched then
      create code object for modifiedClass
      set version of code object to (max_version + 1)
      set version predecessor pointer to vers
    end if
  else
    use new operator to create code object
  end if
```

The `Delete` operation is used to remove a version represented by a code object from the database. If the code object being removed is referred to by other code object, then the Delete operation wont be successful.

6 Evaluation

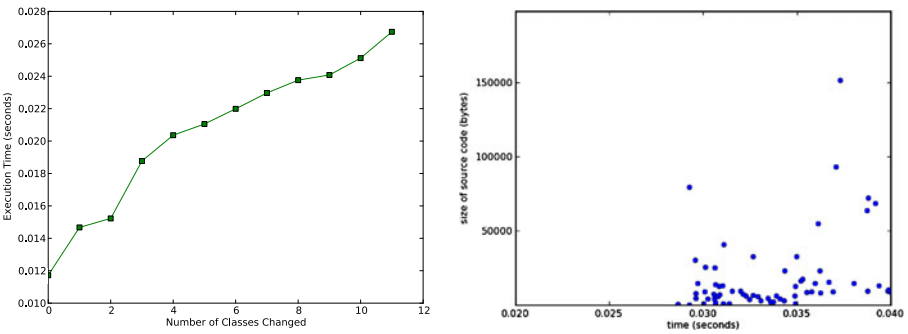
To evaluate our framework, we implemented the algorithms described here in Astro-WISE and we performed studies on real code and objects. In this section, we check the efficiency of the algorithm by observing the time it takes to compute differences between two classes, we also describe a few implementation details.

6.1 Execution time

For the timing experiments, we used a dedicated Intel dual core 2.8Ghz desktop, with 4GB of memory, running GNU/Linux 2.6.18-238.12.1.e15, connected on 1Gbit network connection to the data server and database. In the first test, we randomly made changes to source code of two classes while running our differencing algorithm. Each test was run 1000 times and we took the average. The results of this test are shown in Fig. 7a. The cost of this test is linear to the number of nodes that have been changed. In the second test, we made the same changes to different classes of different sizes. The results are shown in Fig. 7b. From the results, the size of the input documents did not change the processing time. Since, this was done on a live network and execution time is highly dependent on the database, this could explain some of the differences in Fig. 7b. We can conclude from the two results that execution time primarily depends on the number of changed nodes.

6.2 Actualization of COVA

We present in this section a few implementation details for the system in relation to change detection, processing data objects and querying code objects. Astro-WISE implements our object versioning and differencing algorithm. The differencing algorithm inputs the original and modified versions of python files, compares them, and outputs detected differences.



(a) Changes Vs Execution time (b) Size of Documents Vs Execution time

Fig. 7 Performance tests

The AWEPIPE environment variable specifies to Astro-WISE where the local personalized checkout is stored. It is the classes in the personalized checkout that are used to process data. During the processing of a target, the specific classes that are being used to make the target are matched against the code object of the same type stored as the database. If changes are detected a new code object will be created with a new version and this will be linked to the derived object, otherwise the object will be linked to the version of the code object that is being used.

6.3 AWE classes

We briefly describe some of the classes and some special methods that have been implemented for this framework.

6.3.1 DBObject

The DBObject is the base class of all persistent objects in Astro-WISE. Python objects whose classes are derived from DBObject can be stored in the database. Such an object can be instantiated on the Python prompt by querying the database on the persistent properties of the object. DBObject defines the primary key (i.e. object_id) of all objects. The DBObject class defines the store() and commit() methods. Both methods make a transient object persistent.

6.3.2 DataObject

The DataObject class is derived from DBObject and is the base class for persistent classes for which data is stored in a file on the data-server. An example of a Persistent Property is filename which is the name of the associated file as it is stored on the data-server.

6.3.3 CodeObject

The CodeObject class is derived from DataObject and is the base class for persistent code objects. This class provides mechanisms for making code objects, setting version numbers and making each code object persistent in the database. The CodeObject class defines new persistent attribute called methods, which is an array of links to instances of the Methods class.

Since CodeObject is inherited from the DataObject class, it inherits the persistent attribute filename. This attribute links a complete source code file to the code object. The filename attribute is used to implement store() and retrieve() methods to transfer files to and from dataservers. Using

regular kind of expressions, we can search and retrieve files that meet a specific criteria.

6.3.4 CodeObjectMetaData

The `CodeObjectMetaData` class is derived from `DBObject` and class that creates and stores metadata for persistent code objects.

6.3.5 Methods

The `Methods` class is derived from `DBObject` and class that creates and stores information about methods. The `Methods` class defines new persistent attributes as follows

- methodName, Name of method
- methodVersion, the version of the method
- definingClass, the class that defines this method
- methodAttributes, Method arguments defined for this method
- description, description of the method
- defaultReturnValues, Baseline Return Values
- bytecode: The bytecodes of the function, for quick comparison.

6.4 Querying versioning information

We have defined a notation that is based on the idea that a class is in some sense equivalent to the set of all its instances. To illustrate the concept, let us give a few examples. Given a persistent class X with persistent property y, then the expression $X.y == 5$ represents the set of all instances x of X, or subclasses of X, for which $x.y == 5$ is true. To obtain these objects the expression needs to be evaluated, which can be done by passing it to the select function, which returns a list of objects satisfying the selection. Given a class X with a descriptor `desc`, a descriptor list `dsc_lst`, and a link `lnk`, then

```
{select(X.desc > 2.0 && X.dsc_lst[2]= 'abc' and
                X.lnk.attr = 5)}
```

will return a list of instances x of X, or subclasses of X, for which x.desc > 2.0 and x.dsc_lst[2] = 'abc' and x.lnk.attr = 5 is true.

For example, in Astro-WISE methods can be selected by executing this query as below;

```
awe> method = CodeObject.Methods.methodName == 'apply_fringe_correction'
awe> method &= CodeObject.Methods.methodVersion == 2
awe> method &= CodeObject.Methods.definingClass == 'ReducedScienceFrame'
```

6.4.1 Target processing and object versioning

To process a target (data object), a researcher begins sending a query to the database for the target. If the target exists, the target is returned, else the processing of the target is initiated. The inputs to the pipeline are either queried from the database if they already exist or created on the fly if the objects are not *uptodate* or do not exist. This is a recursive process that begins from the target to the raw data as observed from the telescope. An object is *uptodate* if the most specific class and its dependencies that were used to make the object have not changed or if any of its dependencies have not been made by a newer version of a class. This is determined by comparing version of code object that was used to make the object and version of the current (latest) code object. Astro-WISE provides a webservice for this check, an example of this comparison is shown in Fig. 8. If a newer version of class exists, it is highlighted in orange.

The object viewer of each object would give details of the changed attributes, an example is shown in Fig. 9. Notice from the figure that the methods, `check_preconditions`, `copy_attributes`, `debias_and_flatfield_frame` were modified from the previous versions and therefore their methodVersion values should have been incremented from previous versions, depending on how many times the method has been modified.

Also note from Fig. 9, that each code object has an attribute called `Methods` which is a link to methods defined, inherited or imported into that class being versioned. For example, in the figure, the method with methodName `select_for_raw` is defined by the `ProcessTarget` class and currently this method has version 1. If the definingClass re-implements this method, the `showChanges()` method will show which classes have been affected by this change. Then a reprocess command can be issued to create new code object

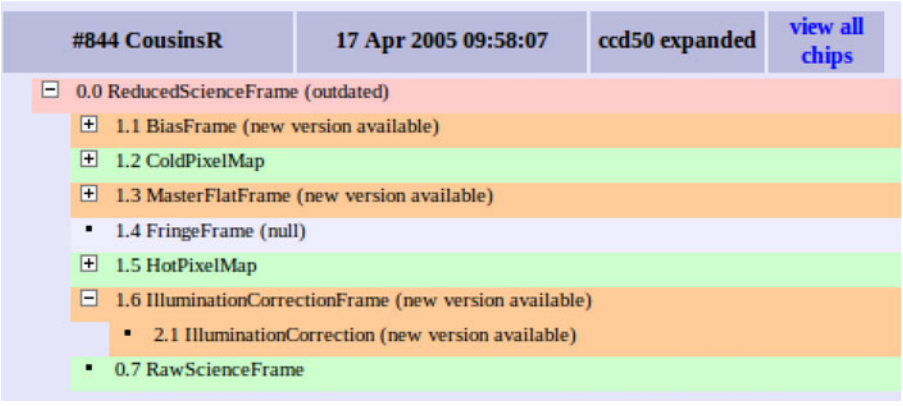


Fig. 8 Dependency graph of classes required to make the `ReducedScienceFrame` object

Fig. 9 The object view of a code objecto of type ReducedScienceFrame

based on the changes in the inherited method. This will increment the version attribute for all the code objects that use the changed method.

For example, using the method `inverse_objects()`, we can retrieve and display all classes that use a particular method, an example is shown in the output below;

```
awe> obj = (CodeObject.Methods.methodName == 'select_for_raw')[0]
awe> klasses = obj.inverse_objects()
awe> for klass in klasses:
... print 'Class: %s, ClassVersion:%d' %(klass.category, klass.code_version)
...
Class: BaseFrame, ClassVersion:1
Class: RegriddedFrame, ClassVersion:1
Class: RawTwilightFlatFrame, ClassVersion:1
Class: ReducedScienceFrame, ClassVersion:1
Class: ReducedScienceFrame, ClassVersion:2
Class: DomeFlatFrame, ClassVersion:2
Class: DomeFlatFrame, ClassVersion:3
Class: ReferenceFrame, ClassVersion:1
Class: SatelliteMap, ClassVersion:1
....
```

7 Related work

Schema evolution and versioning [12] are other related concepts that deal with changes to classes. In order to avoid the loss of data after schema changes, many object-oriented systems use schema evolution, which provides (partial) automatic recovery of the extant data by adapting them to the new schema. However, if only the updated schema is retained, all the applications compiled with the past schema may cease to work. In order to let applications work on multiple schemata, schema versioning [18] is used. Schema versioning makes it possible to view the data under different versions of the schema. However schema versioning often does not take the versioning of instance data into account. Changing classes sometimes may not lead to a schema change but might change results or the process. Such changes are not addressed by schema evolution and versioning mechanisms.

There are a number of existing techniques for computing differences between two versions that also recognize differences in object-oriented features. Semantic diff [11], compares two versions of a program procedure-by-procedure. However, it does not consider dependencies relationship between classes and variables. BMAT [21] performs matching on both code and data blocks between two versions of a program in binary format, however it does not provide information about differences between matched entities. JDIFF [2], uses the OOP approach when comparing classes. Its main focus is to determine differences that change the behavior of a program, e.g., changing branch conditions. Our focus is not behavior of the program, but changes that have an effect on the results of a program. There is also considerable work related to VCSs [16, 17], however their focus is dedicated on modeling software artifacts and therefore version numbers created by VCSs are opaque identifiers and as such can not be used for object versioning.

Insight about storing source code in some intermediate representation (e.g. a relational database) is given in [10]. Source Code in Database (SCID) is a related concept where code is pre-parsed and stored in a database. This leads to the elimination of source code files since source code is stored exclusively in the database. SCIDs has been explored for the purposes of developing Integrated Development Environments (IDE). We use the same concept. However, rather than storing the complete source code, only useful (querible) attributes are extracted and stored in the database while we maintain a link to the source code file which is identified with a uniquely generated name. The advantages are enormous. These range from versioning abilities, linking and performance of the system.

8 Conclusion

The major goal of this work is to support scientific collaboration in a fully distributed way. This requires a shift from traditional data processing to a more

interactive environment which enables scientists collectively reuse, change and adapt scripts developed by their peers to derive new insight or to evaluate their specific hypotheses.

In distributed scientific environment, scientists may work at the same time on source code, change methods (or an implementation) or may even disagree on some implementations. In a such an environment, not all these changes would become part of the source code repository. The work done in this paper, allows such kind of changes to be stored while linking them to the objects they created. All variations of implementations created during the scientific processes become publicly available and can be used to reprocess data, to find data through code-based searches and to understand the process that leads to the creation of a data item. The differencing algorithm for comparing two `Python` programs is based on `Python`'s object-oriented features. The algorithms matches two classes and all its dependencies and identifies differences that would have an effect to the data.

This framework allows potability of data, since format interoperability problems do not exist. Each derived object knows specifically how it was made. Through COVA, the specific class (and/or source code) is known to the object, likewise the processing parameters which are maintained by data provenance can also be retrieved.

However, this method is not full proof and might fail in some cases. Changes can be highly subjective. To know what constitutes a change or more general a major change to necessitate a new version change is still a limitation. In some cases, applying a change can lead to ambiguity and derive multiple similar results. However,we expect these kind of errors to be detected when comparing output of two methods.

Pipelines often consist of a mixer of languages. Entire versioning would require that this framework considers all programming languages. However since `Python` is Astro-WISE's data-definition and modfication langague, we try to solve any changes required to be made in other languages at the python layer, and as such such changes can be versioned.

For this method to be effective, users have to follow the standard object data model so that the changes included do not break the existing data model. For example, anything considered to be a constant is not included in source code, but rather is stored as a processing parameter. This avoids creation of a version where someone has specifically made a change to a constant variable. Secondly, a new-version might be created as a result of changing the order of operations thus producing new versions. If changing the order of operations does not change the data model and hence the processed results, Then such changes will be picked up by the data comparison test (Section 4.1.2).

There are many avenues for future work, we propose to extend this work into a full-fledged entity-based versioning system. This will allow versioning of instance data in much finer-grained ways by eliminating costly parsing and matching steps. Hence data objects can be queried directly without needing the building and definition of code objects.

References

1. Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., Mock, S.: Kepler: an extensible system for design and execution of scientific workflows. In: 16th International Conference on Scientific and Statistical Database Management (SSDBM'04), pp. 423 (2004)
2. Apiwattanapong, T., Orso, A., Harrold, M.J.: A differencing algorithm for object-oriented programs. In: Proceedings of the 19th IEEE International Conference on Automated Software Engineering, pp. 2–13. IEEE Computer Society, Washington, DC, USA (2004)
3. Begeman, K.G., Belikov, A.N., Boxhoorn, D.R., Dijkstra, F., Valentijn, E.A., Vriend, W.J., Zhao, Z.: Merging grid technologies. Journal of Grid Computing **8**, 199– 221 (2010)
4. Cohen-Boulakia, S., Biton, O., Cohen, S., Davidson, S.: Addressing the provenance challenge using zoom. Concurr Comp-Pract E **20**(5), 497–506 (2008)
5. Conte, D., Foggia, P., Sansone, C., Vento, M.: Thirty years of graph matching in pattern recognition. IJPRAI pp. 265–298 (2004)
6. Deelman, E., Singh, G., Su, M.H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G.B., Good, J., Laity, A., Jacob, J.C., Katz, D.S.: Pegasus: a framework for mapping complex scientific workflows onto distributed systems. Sci. Program. **13**, 219–237 (2005). http://dl.acm.org/citation.cfm?id=1239649.1239653
7. Elaine Angelino, D.Y., Seltzer, M.: Starflow: a script-centric data analysis environment. In: McGuinness, D.L., Michaelis, J.R., Moreau, L. (eds.) Provenance and Annotation of Data and Processes, Third International Provenance and Annotation Workshop, IPAW 2010, Troy, NY, USA, 15–16 June 2010. Proceedings, Lecture Notes in Computer Science, vol. 6378. Springer (2010). doi:10.1007/978-3-642-17819-1
8. Foster, I., Vöckler, J., Wilde, M., Zhao, Y.: Chimera: a virtual data system for representing, querying, and automating data derivation. In: 14th International Conference on Scientific and Statistical Database Management (SSDBM'02), pp. 37 (2002)
9. Freire, J., Koop, D., Santos, E., Silva, C.T.: Provenance for computational tasks: a survey. Comput. Sci. Eng. **10**(3), 11–21 (2008)
10. Green, R.: Sorce code in databases. Java Source Code SCID-style browser/editor. http://mindprod.com/project/scid.html (2011)
11. Jackson, D., Ladd, D.A.: Semantic diff: a tool for summarizing the effects of modifications. In: Proceedings of the International Conference on Software Maintenance, ICSM '94, pp. 243–252. IEEE Computer Society, Washington, DC, USA (1994)
12. Jørgensen, P.S., Böhlen, M.: Versioned relations: support for conditional schema changes and schema versioning. In: Proceedings of the 12th International Conference on Database Systems for Advanced Applications, DASFAA'07, pp. 1058–1061. Springer, Berlin, Heidelberg (2007)
13. Myers, E.W.: An o(nd) difference algorithm and its variations. Algorithmica **1**, 251–266 (1986)
14. Ogasawara, E., Rangel, P., Murta, L., Werner, C., Mattoso, M.: Comparison and versioning of scientific workflows. In: Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models, CVSM '09, pp. 25–30. IEEE Computer Society, Washington, DC, USA (2009)
15. Oliveira, F.T., Murta, L., Werner, C., Mattoso, M.: Using provenance to improve workflow design. In: Freire, J., Koop, D., Moreau, L. (eds.) Provenance and Annotation of Data and Processes, pp. 136–143. Springer, Berlin, Heidelberg (2008)
16. Oliveira, H., Murta, L., Werner, C.: Odyssey-vcs: a flexible version control system for uml model elements. In: Proceedings of the 12th International Workshop on Software Configuration Management, SCM '05, pp. 1–16. ACM, New York, NY, USA (2005)

17. Robbes, R., Lanza, M.: Versioning systems for evolution research. In: Proceedings of the Eighth International Workshop on Principles of Software Evolution, pp. 155–164. IEEE Computer Society, Washington, DC, USA (2005). doi:10.1109/IWPSE.2005.32. http://portal.acm.org/citation.cfm?id=1107840.1108157
18. Roddick, J.F.: A survey of schema versioning issues for database systems. Inf. Softw. Technol. **37**, 383–393 (1995)
19. Simmhan, Y., Plale, B., Gannon, D.: Karma2: provenance management for data-driven workflows. IJWSR **5**(2), 1–22 (2008)
20. Valentijn, E.A., McFarland, J., Snigula, J., Begeman, K., Boxhoorn, D., Renegelink, R., Helmich, E., Heraudeau, P., Kleijn, G.V., Vermeij, R., Vriend, W.J., Tempelaar, M.J.: Astro-wise: chaining to the universe. In: Astronomical Data Analysis Software and Systems XVI, ASP Conference Series, vol. 376 (2007)
21. Wang, Z., Pierce, K., McFarling, S.: Bmat–a binary matching tool for stale profile propagation. JILP **2**, 2000 (2002)